Aug 25th, 2025

Revision 1.0

Superform V2 Core

Prepared by Orion

contact@orionsecurity.xyz

Table of Contents

Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where our team tries to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

1. Introduction

Orion performed a targeted security assessment of Superform's V2 core contracts. The review aimed to uncover potential risks in the contract logic, assess the resilience of the system under common attack vectors, and provide actionable recommendations to improve the protocol's overall security. The engagement was conducted over a period of three weeks: during the first two weeks, two researchers collaborated on the review, followed by a final week of work carried out by a single researcher.

1.1. Executive summary

A total of twenty-seven (27) findings were identified, including seven (7) medium-severity issues. No critical or high issues were found.

While the codebase has been subject to prior security audits, including a Cantina competition, recent updates introduced several new features and modifications that warranted focused review.

Key areas of attention included full EIP-1271 compliance for contract signers (including Safe multi-sigs) and associated chain abstraction mechanisms, integration with ERC-7579, and enhancements to Safe multi-sig support. We also evaluated new execution paths introduced by EIP-7702, the implementation of BaseHook with DoS prevention, and mechanisms such as SuperSenderCreator, InvalidateMerkleRoot, and the MerkleClaim and Circle Gateway hooks.

Additionally, we verified standards conformance across Executors, Paymasters, and Validators, ensuring correct handling of pre- and post-execution operations.

Our review further focused on accounting and operational correctness, including potential rounding issues, fee skipping, and DoS vectors, as well as cross-chain edge cases that could result in unexpected refunds.

The following sections present comprehensive technical details and remediation guidance for all identified issues.

1.2 Scope

Repository	superform-xyz/v2-core/
Commit	@25d847c6

The audit covered the following files:

Category	Files / Directories
Core	`src/accounting/` `src/adapters/` `src/executors/` `src/paymaster/` `src/validators/*`
Interfaces & Libraries	`src/interfaces/` `src/libraries/`
Hooks	`src/core/hooks/bridges/` `src/hooks/claim/merkl/MerklClaimRewardHook.sol` `src/hooks/swappers/1inch/` `src/hooks/swappers/odos/` `src/hooks/superform/MarkRootAsUsedHook.sol` `src/hooks/tokens/` `src/hooks/vaults/4626/` `src/hooks/vaults/5115/` `src/hooks/vaults/7540/` `src/hooks/vaults/ethena/` `src/hooks/BaseHook.sol`
Scripts	`script/DeployV2Base.s.sol` `script/DeployV2Core.s.sol` `script/utils/*`

In addition to the initial scope, the following contracts were introduced in PR 781 during the audit, and were also reviewed by our team:

- `src/core/hooks/bridges/CircleGatewayAddDelegateHook.sol`
- `src/core/hooks/bridges/CircleGatewayRemoveDelegateHook.sol`

1.3 Overview of findings

Our comprehensive review identified twenty-seven (27) findings, seven (7) of which were classified as medium severity:

Severity	Count
HIGH	0
MEDIUM	7
LOW	5
INFO	15
TOTAL	27

2. Findings

M-01: Chain-agnostic validation doesn’t support all Safe-supported signature types MEDIUM

Summary

The `ChainAgnosticSafeSignatureValidation` library doesn’t support all kinds of valid Safe signatures, only allowing Safes where owners are EOA’s.

Vulnerability details

When a signature is processed, Superform will try to identify if it belongs to a Safe, and check wether enough Safe’s owners have signed so that the threshold is reached. This is done via the `ChainAgnosticSafeSignatureValidation`’s `validateChainAgnosticMultisig` function. After computing the `chainAgnosticHash`, the internal `_verifyMultisigSignatures` function will be called to perform the signature verification:

```
// File: ChainAgnosticSafeSignatureValidation.sol

function _verifyMultisigSignatures(
        bytes32 messageHash,
        bytes memory signatures,
        address[] memory owners,
        uint256 threshold
    )
        private
        pure
        returns (bool)
    {

        ...SNIP

        for (uint256 i = 0; i < threshold; i++) {
            ...SNIP

            // Use OpenZeppelin's secure ECDSA recovery with proper validation
            // tryRecover performs all security checks: s-value malleability, v
        validation, etc.
            address currentOwner = ECDSA.recover(messageHash, currentSignature);

            // Check if recovered address is a valid owner and maintains ascending order
            if (_isOwner(currentOwner, owners)) {
                // Check ordering - must be ascending to prevent signature reuse
                if (currentOwner <= lastOwner) {
                    return false; // Signatures not in ascending order
                }

                validSignatures++;
                lastOwner = currentOwner;
            }
        }

        // All signatures must be valid and from distinct owners
        return validSignatures == threshold;
    }
```

The problem is that only ECDSA is used in order to determine the signer of the `messageHash`, while Safe supports a wide range of signatures. This prevents certain Safe setups, for example where an owner of the Safe might be an actual contract, and an EIP-1271 signature validation should be performed, instead of ECDSA.

Impact

Medium, some Safe setups won't be supported, making it impossible for certain users to interact with the protocol.

Recommendation

Consider updating the signature verification approach for Safes so that all of Safe's supported signature types are checked.

Superform

Fixed in PR 743

Orion

Fix confirmed. An approach similar to Rhinestone's `checknsignatures` is used, where all kinds of Safe-supported signatures will be checked.

M-02: Some executions for user operations with multiple target chains can be DoS'ed MEDIUM

Summary

Insufficient validation makes it possible to DoS some destination executions if more than one destination is targeted.

Vulnerability details

When a user operation is submitted with a cross-chain execution, the `validateUserOp` will re-compute the destination leaf and validate it is part of the submitted `merkleRoot`:

```
// File: SuperValidator.sol
function validateUserOp(
        PackedUserOperation calldata _userOp,
        bytes32 _userOpHash
    )
        external
        override
        returns (ValidationData)
    {
        ...SNIP

        // Verify destination data
        if (isValid && sigData.validateDstProof) {
            uint256 dstLen = sigData.proofDst.length;
            if (dstLen == 0) revert INVALID_DESTINATION_PROOF();

            for (uint256 i; i < dstLen; ++i) {
                DstProof memory dstProof = sigData.proofDst[i];
                // for which we ensure inclusion in the merkle tree
                DestinationData memory dstData = DestinationData({
                    callData: dstProof.info.data,
                    chainId: dstProof.dstChainId,
                    sender: dstProof.info.account,
                    executor: dstProof.info.executor,
                    dstTokens: dstProof.info.dstTokens,
                    intentAmounts: dstProof.info.intentAmounts
                });

                // Create destination leaf
                bytes32 dstLeaf = _createDestinationLeaf(dstData, sigData.validUntil,
        dstProof.info.validator);

                // Ensure leaf belongs to signed root
                if (!MerkleProof.verify(dstProof.proof, sigData.merkleRoot, dstLeaf)) {
                    revert INVALID_DESTINATION_PROOF();
                }
            }
```

The signature data incorporates a `validateDstProof` flag which forces validation to be performed for the destination proofs. The problem is that it is never ensured that all destination proofs have been submitted, given that proofs are not part of the merkle tree (and hence can be tampered with until a certain degree).

Let's consider a scenario where a user wants to send a message from ETH → BASE and also ETH → OP. Given that `sigData` is not hashed, there's two scenarios where an attacker can force only one message to succeed:

1. Attacker provides `proofDst.length` of 1, instead of 2, only with the `proofDst` for the ETH → BASE transfer. The destination leaf is valid, and the merkle proof is verified. However nothing enforces that another `proofDst` is submitted, so the validation passes and the ETH → OP proof is never supplied, which will make the execution on destination fail.
2. Attacker provides two `proofDst`, but they're the same (for example, the ETH → BASE proof). The validation still passes, however on OP destination, a valid proof is never found, thus DoS'ing the execution.

Impact

Medium. For all user operations which target more than one destination chain, `n - 1` destination executions can be dos'ed (where `n` is the total destination executions).

Recommendation

Consider enforcing validation for all the destination chains that the signed execution expects to interact with.

Superform

Fixed in PR 755.

Orion

Fix confirmed. The fix updates `validateDstProof` to become `chainsWithDestinationExecution`, an array of destination chain ID's, effectively preventing attackers from supplying incorrect proofs.

M-03: Current signature processing logic will lead to DoS for Safes in certain scenarios MEDIUM

Summary

The logic in `_processSignatureAndVerifyLeaf` will check if the `owner` is an EIP-1271 signer in order to perform a Safe verification or an EIP-1271 verification. However, Safes might not always support EIP-1271, which will prevent chain agnostic validations to be performed, even if valid signatures were supplied.

Vulnerability details

In `_processSignatureAndVerifyLeaf`, `_processEIP1271Signature` will only be called if the `owner` supports EIP-1721:

```
// File: SuperValidator.sol
function _processSignatureAndVerifyLeaf(
        address sender,
        SignatureData memory sigData,
        bytes32 userOpHash
    )
        private
        view
        returns (address signer, bytes32 leaf)
    {
        ...SNIP
        if (_is7702Account(sender.code)) {
             ...SNIP
        } else {
            address owner = _accountOwners[sender];

            // Support any EIP-1271 compatible smart contract, not just Safe multisigs
            if (_isEIP1271Signer(owner)) {
                signer = _processEIP1271Signature(owner, sigData);
            } else {
               ...SNIP

          }
  }
```

`_processEIP1271Signature` will carry out signature verifications for both Safe and EIP-1271-compliant contracts. The problem is that Safe's will only support EIP-1271 if they have configured the CompatibilityFallbackHandler. There's other fallback handlers though, which leads us to the following potential scenario:

1. We have a Safe that has activated the TokenCallbackHandler. Because of this, the Safe can't have the CompatibilityFallbackHandler active.
2. The Safe's owners have signed the chain-agnostic signature. The signatures should be validated through the `validateChainAgnosticMultisig` function.
3. When `_processSignatureAndVerifyLeaf` is called, the internal `_isEIP1271Signer` is used to determine wether the owner is an EIP1271 signer. However this will return false, because the expected fallback handler is not configured, so it won't be possible to accept accounts that have this kind of setup with Safes, given that the current logic will incorrectly assume that the owner is an EOA.

Impact

Medium. This issue will prevent certain Safes from using the protocol, even if they have provided valid signatures from their owners so that chain agnostic validation is performed.

Recommendation

Consider updating the logic in `_processSignatureAndVerifyLeaf` so that signatures provided for Safes not supporting EIP-1271 are still allowed to be verified via chain-agnostic validation.

Superform

Fixed in PR 746.

Orion

Fix confirmed. The logic to identify the type of account has been updated to avoid the mentioned DoS scenario.

M-04: Users can bypass fees by leveraging malicious logic used in the account's `withHook` modifier MEDIUM

Summary

Users can add custom logic to their accounts and leverage the `withHook` modifier, configuring a malicious hook to bypass all fees.

Vulnerability details

As part of issue 436 found during the Cantina audit competition, some changes were introduced to the execution logic so that the attack to avoid fees leveraging the `withHook` modifier could be mitigated. The main fixes consisted in introducing mutexes, flags that configure temporary transient-based access control to limit the account's capabilities during a hook execution.

Although the fix mitigated most of the attacks, there's still a way to bypass fees: 1. Initially, the `SuperExecutor` will call `setExecutionContext`. This sets a context for the account, and also configures `lastCaller` to be the `SuperExecutor`. 2. Hook functions are called (`preExecute` → custom hook executions → `postExecute`). At this point: - There's an execution context for the account - Mutexes for pre/post executions have been set - Final `outAmount` has been computed 3. Before finalizing the account's execution, this part of `withHook` is called. The nexus hook has been set as `executor` for the account, so `executeFromExecutor` is called in the account. The execution calls these: - `resetExecutionState` - `preExecute` and `postExecute`. This makes `outAmount` become 0. Note that this step could also be performed by calling `setOutAmount`, and the final result would be the same (final `amountOut` being zero). 4. `executeFromExecutor` finalizes, and the regular SuperForm flow continues. The `SuperExecutor` calls `resetExecutionState`, and finally fees are computed, which become 0.

Impact

Medium. Although mitigations were added, a similar attack to issue 436 can be triggered, effectively avoiding paying for any fees. Given that there's other ways to bypass fees in the protocol and this is considered a known risk, medium severity is appropriate for this issue.

Proof of concept

The following proof of concept illustrates the issue. A malicious hook is created and configured as an executor for the account. Then, the corresponding calldata is configured so that the hook calls `resetExecutionState` → `preExecute` → `postExecute` again, so that the fees become zero.

```
// File: E2EExecution.t.sol

function test_feeBypassByResettingExecution() public {
        uint256 amount = 10_000e6;
        address underlyingToken = CHAIN_1_USDC;
        address morphoVault = CHAIN_1_MorphoVault;

        MaliciousHookResetExecution maliciousHookResetExecution = new
         MaliciousHookResetExecution();

        // Step 1: Create account and install custom malicious hook
        address nexusAccount = _createWithNexusWithMaliciousHook(
            attesters,
            threshold,
            1e18,
            address(maliciousHookResetExecution)
        );

        maliciousHookResetExecution.setAccountAndTargetHook(
            nexusAccount,
            redeem4626Hook
        );

        // add tokens to account
        _getTokens(underlyingToken, nexusAccount, amount);

        // Step 2: Install hook as an account executor of the account
        vm.prank(nexusAccount);
        INexus(nexusAccount).installModule(
            2,
            address(maliciousHookResetExecution),
            ""
        );

        // Configure malicious executions in the hook
        Execution[] memory executions = new Execution[](3);


        executions[0] = Execution({
            target: address(redeem4626Hook),
            value: 0,
            callData: abi.encodeWithSignature(
                "resetExecutionState(address)",
                nexusAccount // account
            )
        });
        bytes memory redeemData = _createRedeem4626HookData(
            _getYieldSourceOracleId(
                bytes32(bytes(ERC4626_YIELD_SOURCE_ORACLE_KEY)),
                address(this)
            ),
            morphoVault,
            nexusAccount,
            IERC4626(morphoVault).convertToShares(amount),
            false
        );
        executions[1] = Execution({
            target: address(redeem4626Hook),
            value: 0,
            callData: abi.encodeWithSignature(
                "preExecute(address,address,bytes)",
                address(0), // prevHook
                nexusAccount,
                redeemData
            )
        });
        executions[2] = Execution({
            target: address(redeem4626Hook),
            value: 0,
            callData: abi.encodeWithSignature(
                "postExecute(address,address,bytes)",
                address(0), // prevHook
                nexusAccount,
                redeemData
            )
        });


        bytes memory data = abi.encodeCall(
            IERC7579Account.executeFromExecutor,
            (ModeLib.encodeSimpleBatch(), ExecutionLib.encodeBatch(executions))
        );

        maliciousHookResetExecution.setTargetCalldata(data);

        // Step 3. Create SuperExecutor data, with:
        // - approval
        // - deposit
        // - redemption, whose amount should be charged

        address[] memory hooksAddresses = new address[](3);
        hooksAddresses[0] = approveHook;
        hooksAddresses[1] = deposit4626Hook;
        hooksAddresses[2] = redeem4626Hook;

        bytes[] memory hooksData = new bytes[](3);
        hooksData[0] = _createApproveHookData(
            underlyingToken,
            morphoVault,
            amount,
            false
        );
        hooksData[1] = _createDeposit4626HookData(
            _getYieldSourceOracleId(
                bytes32(bytes(ERC4626_YIELD_SOURCE_ORACLE_KEY)),
                address(this)
            ),
            morphoVault,
            amount,
            false,
            address(0),
            0
        );
        hooksData[2] = _createRedeem4626HookData(
            _getYieldSourceOracleId(
                bytes32(bytes(ERC4626_YIELD_SOURCE_ORACLE_KEY)),
                address(this)
            ),
            morphoVault,
            nexusAccount,
            IERC4626(morphoVault).convertToShares(amount),
            false
        );

        // Step 4. Prepare data and execute through entry point
        ISuperExecutor.ExecutorEntry memory entry = ISuperExecutor
            .ExecutorEntry({
                hooksAddresses: hooksAddresses,
                hooksData: hooksData
            });

        address feeRecipient = makeAddr("feeRecipient"); // this is the recipient
        configured in base tests.

        // Fetch the fee recipient balance before execution
        uint256 feeReceiverBalanceBefore = IERC4626(CHAIN_1_USDC).balanceOf(
            feeRecipient
        );

        _executeThroughEntrypoint(nexusAccount, entry);

        // Ensure fee is 0
        assertEq(
            IERC4626(CHAIN_1_USDC).balanceOf(feeRecipient) -
                feeReceiverBalanceBefore,
            0
        );
    }
```

The malicious hook is a simple contract that will execute the configured calldata after a certain number of calls:

```solidity
// File: MaliciousHookResetExecution.sol

// SPDX-License-Identifier: MIT
pragma solidity 0.8.30;

contract MaliciousHookResetExecution {
    address public account;
    address public targetHook;
    uint256 public counter;

    uint256 constant EXECUTOR_TYPE_HOOK = 2;
    uint256 constant MODULE_TYPE_HOOK = 4;

    bytes public data;

    function setAccountAndTargetHook(
        address _account,
        address _targetHook
    ) external {
        account = _account;
        targetHook = _targetHook;
    }

    function setTargetCalldata(bytes memory _data) external {
        data = _data;
    }

    function preCheck(
        address msgSender,
        uint256 msgValue,
        bytes calldata msgData
    ) external returns (bytes memory hookData) {
        // do nothing in precheck
    }

    function postCheck(bytes calldata /*hookData*/) external {
        // Call the account
        ++counter;
        if (counter == 4) {

            (bool success, ) = account.call(data);

            if (!success) revert("Failed to trigger attack");
        }
    }

    function isModuleType(uint256 moduleTypeID) external pure returns (bool) {
        return
            moduleTypeID == MODULE_TYPE_HOOK ||
            moduleTypeID == EXECUTOR_TYPE_HOOK;
    }
    function onInstall(bytes calldata data) external {}
}
```

The poc can be executed by running `forge test --mt test_feeBypassByResettingExecution -vvvv`. The execution trace shows the attack process described in this issue, and the poc ensures the total amount of assets received by the fee receiver is 0.

Recommendation

Consider enforcing `resetExecutionState` so that it can only be called by the configured `lastCaller`. This will make it impossible for the account to change the state of the mutexes, effectively preventing the attack from taking place.

Superform

Fixed in PR 754.

Orion

Fix confirmed. An `onlyLastCaller` modifier was introduced, which makes `resetExecutionState` only be callable by `lastCaller`.

M-05: `CircleGatewayMinterHook` does not support `AttestationSet` MEDIUM

Summary

Attestation sets are a first-class input to Circle's minter. The hook does not support `AttestationSet` and assumes a single `Attestation`. Since users are likely to aggregate multiple burns (e.g., USDC from multiple source chains) into one `AttestationSet`, valid inputs will revert.

Vulnerability details

- Circle flow supports `AttestationSet`:
    - `gatewayMint` accepts a single `Attestation` or an `AttestationSet` and iterates each element:

https://github.com/circlefin/evm-gateway-contracts/blob/master/src/modules/minter/Mints.sol#L161-L193

```
function gatewayMint(bytes memory attestationPayload, bytes memory signature) external
        whenNotPaused notDenylisted(msg.sender) {
    ...SNIP
    Cursor memory cursor = AttestationLib.cursor(attestationPayload);
    ...SNIP
    while (!cursor.done) {
        attestation = cursor.next();
        uint32 index = cursor.index - 1;
        _validateAttestationNotExpired(attestation, index);
        bytes29 spec = attestation.getTransferSpec();
        _validateAttestationTransferSpec(spec, index);
        _mint(spec);
    }
}
```

- On the other hand `CircleGatewayMinterHook` does not handle `AttestationSet`:
    - `_extractTokenFromAttestation` calls `AttestationLib._validate(attestationPayload)` and then `getTransferSpec` on the returned view, assuming a single `Attestation`. If the payload is an `AttestationSet`, this reverts.

Without casting to a `Cursor` and iterating, valid signed `AttestationSet`s will not work and revert in `_extractTokenFromAttestation()`.

Impact

Medium. Valid signed `AttestationSet`s will cause the hook to revert, leading to failed executions and poor UX for a common use case (multi-chain USDC burns aggregated into a single mint on the destination chain).

Recommendation

- Add `AttestationSet` support in `CircleGatewayMinterHook.sol`:
    - Use `AttestationLib.cursor(attestationPayload)` to iterate all entries in the `AttestationSet`.
    - Ensure the hook does not revert when given a valid `AttestationSet`.

Superform

Fixed in PR 751.

Orion

Fix confirmed.

M-06: `AttestationSet` may include multiple tokens MEDIUM

Summary

`gatewayMint` can process a single `Attestation` or an `AttestationSet`, and may mint multiple tokens in one call when an attestation set is used.

The hook `CircleGatewayMinterHook.sol` extracts a single token from the payload and does not validate that all attestations in the set reference the same token.

Vulnerability details

`gatewayMint` iterates over all attestations and mints for each element in the set, without enforcing that all `destinationToken` values are equal:

https://github.com/circlefin/evm-gateway-contracts/blob/master/src/modules/minter/Mints.sol#L177-L192

```
// Iterate over the attestations, validating and processing each one
bytes29 attestation;
while (!cursor.done) {
    attestation = cursor.next();

    ...SNIP

    bytes29 spec = attestation.getTransferSpec();
    _validateAttestationTransferSpec(spec, index);

    // Mint funds according to the spec
    _mint(spec);
}
```

In contrast, the hook validates/extracts only a single token from the payload and does not iterate the `AttestationSet` to check consistency:

```
    function inspect(bytes calldata data) external pure override returns (bytes memory)
        {
        address usdc = _extractTokenFromAttestation(data);
        return abi.encodePacked(usdc);
    }
```

```
function _extractTokenFromAttestation(bytes memory data) internal pure returns (address
        usdc) {
    ...SNIP
    bytes32 usdcBytes32 = TransferSpecLib.getDestinationToken(transferSpec);
    usdc = AddressLib._bytes32ToAddress(usdcBytes32); // should iterate over attestation
        set and enforce a single token
```

Consequences: - `inspect()` returns a single token, potentially misrepresenting a set containing multiple tokens. - `_preExecute/_postExecute` compute balances for one token only, while the Circle call may mint additional tokens. Even though this is a NONACCOUNTING hook, downstream logic relying on `asset` or `outAmount` can be inconsistent.

Impact

Medium. The hook may accept and execute an attestation set that mints multiple different tokens while only validating/tracking one. This can lead to incorrect assumptions about the destination asset and minted amount, breaking invariants in subsequent hooks or off-chain logic relying on `inspect()` / `outAmount`.

Recommendation

- In `CircleGatewayMinterHook.sol`, parse the attestation payload via a cursor and iterate all attestations in the set. Extract each `destinationToken` and require that all are identical; otherwise revert with a clear error.

Superform

Fixed in PR 751.

Orion

Fix confirmed.

M-07: Cached `externalCallAdapter` can become stale leading to stuck orders MEDIUM

Summary

The `DebridgeAdapter` caches the `externalCallAdapter` address during construction, but this address can be updated in the `dlnDestination` contract. When the cached address becomes stale, the adapter becomes unreachable, potentially leading to stuck orders that cannot be executed or cancelled.

Vulnerability details

The `DebridgeAdapter` stores the `externalCallAdapter` address as an immutable variable during construction:

```
constructor(address dlnDestination, address superDestinationExecutor_) {
    address _externalCallAdapter =
        IDlnDestination(dlnDestination).externalCallAdapter();
    externalCallAdapter = _externalCallAdapter; // Cached at deployment time
}

modifier onlyExternalCallAdapter() {
    if (msg.sender != externalCallAdapter) revert ONLY_EXTERNAL_CALL_ADAPTER();
    _;
}
```

However, the `externalCallAdapter` can be updated in the `dlnDestination` contract after the `DebridgeAdapter` is deployed. When this happens:

1. **Execution becomes impossible**: The new `externalCallAdapter` will call the `DebridgeAdapter`, but the `onlyExternalCallAdapter` modifier will revert because `msg.sender` no longer matches the cached address.

2. **Orders become stuck**: If an order doesn't have `orderAuthorityAddressDst` set, it cannot be cancelled using `sendEvmOrderCancel()`, effectively trapping the funds.

3. **Deployment dependency**: The entire adapter contract would need to be redeployed with the new `externalCallAdapter` address, breaking existing integrations.

The vulnerability is particularly severe when: - The `externalCallAdapter` is updated during order execution - Orders lack `orderAuthorityAddressDst` (no cancellation authority) - Multiple orders are in-flight when the update occurs

Impact

Medium. Orders can become permanently stuck when `externalCallAdapter` is updated and no `orderAuthorityAddressDst` is set. This leads to: - Permanent fund lock for orders without cancellation authority - Complete service disruption until contract redeployment - Loss of in-flight transactions during the transition period

Recommendation

Store the `dlnDestination` address and dynamically fetch the current `externalCallAdapter`:

```
address public immutable dlnDestination;

constructor(address _dlnDestination, address superDestinationExecutor_) {
    dlnDestination = _dlnDestination;
    // Remove externalCallAdapter caching
}

modifier onlyExternalCallAdapter() {
    if (msg.sender != IDlnDestination(dlnDestination).externalCallAdapter()) {
        revert ONLY_EXTERNAL_CALL_ADAPTER();
    }
    _;
}
```

Superform

Fixed in PR 779.

Orion

Fix confirmed. The contract now stores the `DLN_DESTINATION` address, instead of the `externalCallAdapter` one. The `onlyExternalCallAdapter` was updated to always fetch the external call adapter from the `dlnDestination` contract.

L-01: ERC-4337's `validAfter` field is not supported LOW

Vulnerability details

ERC-4337 allows a `validAfter` field to be present in the returned `validationData` from the `validateUserOp` call. From the EIP: *"`validAfter` is 6-byte timestamp. The UserOperation is valid only after this time."*

However, Superform always hardcodes this value to 0 in the return data from the `validateUserOp` call:

```
// File: SuperValidator.sol

function validateUserOp(
        PackedUserOperation calldata _userOp,
        bytes32 _userOpHash
    )
        external
        override
        returns (ValidationData)
    {
        ...SNIP

        return _packValidationData(!isValid, sigData.validUntil, 0); <@
    }
```

This makes it impossible for users to sign userOps with an actual `validAfter` field, breaking compatibility with the EIP.

Impact

Low. Users won't be able to set a `validAfter` field, which essentially means that signatures will always be valid from the time they are signed, and until the specified `validUntil` is reached.

Recommendation

Consider allowing users to supply a custom `validAfter` field for their userOps.

Superform

Fixed in PR 756.

Orion

Fix confirmed. The PR adds changes to the source leaf so that a `validAfter` timestamp can be successfully passed by users without being manipulatable.

L-02: `postOp` gas is passed as an arbitrary parameter by accounts, allowing users to obtain a higher refund than expected LOW

Vulnerability details

A fix was introduced after Cantina's issue 192 so that the `postOp` gas amount is considered when computing refunds for users utilizing the Paymaster to sponsor their executions. The fix allowed users to add a `postOpGas` field in the encoded `paymasterAndData` field from the userOp:

```
// File: SuperNativePaymaster.sol

function _validatePaymasterUserOp(
        PackedUserOperation calldata userOp,
        bytes32,
        uint256
    )
        internal
        virtual
        override
        returns (bytes memory context, uint256 validationData)
    {
        (uint256 maxGasLimit, uint256 nodeOperatorPremium, uint256 postOpGas) = <@
            abi.decode(userOp.paymasterAndData[PAYMASTER_DATA_OFFSET:], (uint256,
        uint256, uint256));

        if (nodeOperatorPremium > MAX_NODE_OPERATOR_PREMIUM) {
            revert INVALID_NODE_OPERATOR_PREMIUM();
        }
        return (abi.encode(userOp.sender, userOp.unpackMaxFeePerGas(), maxGasLimit,
        nodeOperatorPremium, postOpGas), 0);
    }
```

Allowing the user to control the `postOpGas` is problematic, given that they can simply pass a zero amount in order to obtain a bigger refund when the `postOp` call is executed.

Impact

Low. The total refund will be small, however it still leads to the protocol refunding more assets than intended.

Recommendation

Consider configuring the `postOpGas` in the paymaster contract, instead of allowing users to arbitrarily pass the value.

Superform

Acknowledged. *"When this issue came up, we had the choice to either make it a fixed value in the contract or send it in `paymasterAndData`. We decided to keep it dynamic, so as to give us the ability to make it more accurate with time and not have to deploy another version of the paymaster. As for someone sending really small `paymasterAndData`, this would lead the bundler loosing out on funds as the refund will be higher. But we acknowledge this and within the bundler, before sending a transaction to handleOps of superNativePaymaster we make sure that this value is within a specified range. This is part of validations that we do in the bundler in lines with EIP-7562 checks we do in the bundler. This would be similar to user sending really small `maxFeePerGas` within `paymasterAndData` to change the refund values, hence something we validate in bundler too."*

Orion

Acknowledged by the Superform team.

L-03: `_postOp` incorrectly adds gas amount to total gas cost LOW

Vulnerability details

In the paymaster's `_postOp` call, the `postOpGas` parameter that was previously encoded in `_validatePaymasterUserOp` is used to compute the final `actualGasCost`. This is the total gas cost used up until now (including the cost of calling `postOp`):

```
// File: SuperNativePaymaster.sol

function _postOp(
        PostOpMode,
        bytes calldata context,
        uint256 actualGasCost,
        uint256
    )
        /**
         * actualUserOpFeePerGas
         */
        internal
        virtual
        override
    {
        (address sender, uint256 maxFeePerGas, uint256 maxGasLimit, uint256
        nodeOperatorPremium, uint256 postOpGas) =
            abi.decode(context, (address, uint256, uint256, uint256, uint256));

        // add postOpGas
        actualGasCost += postOpGas;
        uint256 refund = calculateRefund(maxGasLimit, maxFeePerGas, actualGasCost,
        nodeOperatorPremium);
      ...SNIP
    }
```

The problem here is that `postOpGas` is a gas amount, while `actualGasCost` is not an amount, but a cost (gas amount * gas price). Not multiplying `postOpGas` by the expected gas price causes `actualGasCost` to be smaller than it should, as two different units are added.

Impact

Low. The final `actualGasCost` will always be lower than expected, leading to refunds being higher than expected. However the loss will be minimal.

Recommendation

Consider deriving the amount to add to the `actualGasCost` by multiplying the `postOpGas` by the userOp's gas price (configured as `maxFeePerGas` in the `userOp`).

Superform

Fixed in PR 759.

Orion

Fix confirmed. `postOpGas` is now multiplied by `maxFeePerGas` to derive the final gas cost to add to `actualGasCost`.

L-04: Circle `gatewayMint` can be frontrun when `destinationCaller == address(0)` LOW

Vulnerability details

The `CircleGatewayMinterHook.sol` hook constructs a single execution that calls `IGatewayMinter.gatewayMint`. If this call is frontrun, it will revert; because hooks execute sequentially, subsequent hooks in the same execution will not run.

This is possible when the attestation's `destinationCaller` is `address(0)`. In that case, Circle's `gatewayMint(attestationPayload, signature)` imposes no caller restriction. Anyone can submit the attestation first; replay protection then causes later submissions (e.g., via a hook) to revert, aborting the remaining hook chain.

Impact

Low. Potential DoS and UX degradation. A frontrun consumes the attestation, the hook reverts, and subsequent hooks do not execute. No fund loss.

Recommendation

- If the attestation is requested via the Superform frontend, require `destinationCaller` to be set to the destination account (the user's smart account) when generating the attestation to bind usage to that account and prevent frontrunning.
- If `destinationCaller != address(0)`, pre-check `destinationCaller` in the hook and require `destinationCaller == account` to fail fast before calling `gatewayMint`.
- If a user submits an attestation with `destinationCaller == address(0)`, allow execution (do not block).

Superform

Fixed in PR 751.

Orion

Fix confirmed.

L-05: Swappers still miss updating slippage protection when `usePrevHookAmount` is set to `true` LOW

Vulnerability description

Issue 364 in the Cantina competition flagged an issue where the slippage parameter was not updated when `usePrevHookAmount` is set to `true`.

While this was fixed in PR 632, the fix is insufficient, given that it only addressed the "unoswap" flow in the 1inch hook. The generic swap flow still misses updating the `minReturnAmount` value, and the Clipper swap flow misses updating the `outputAmount`.

In addition, the issue is also present in the `SwapOdosV2Hook` and `ApproveAndSwapOdosV2Hook`, given that the `outputMin` parameter is also never updated when `usePrevHookAmount` is `true`.

Impact

Low. This can lead to some unexpected reverts, as the minimum expected output can be too high considering the expected amount that will be finally swapped if `usePrevHookAmount` is set to `true`.

Recommendation

Consider updating the `Swap1InchHook`, `ApproveAndSwapOdosV2Hook` and `SwapOdosV2Hook` so that all the slippage parameters account for the difference between the initial amount and the final amount obtained from the previous hook.

Superform

Fixed in PR 778 and PR 780.

Orion

Fix confirmed. The `HookDataUpdater.sol` library is introduced, which handles the output amount logic update when the previous hook's amount is used. PR 778 introduced a bug making the increase/decrease always be set to 100% when `_prevAmount` is 0. This bug was mitigated in PR 780.

I-01: `_processEIP1271Signature` can be optimized to avoid redundant calls INFO

Vulnerability details

In `_processEIP1271Signature`, the internal `_createMessageHash` is called twice: once to validate the chain agnostic signature, and another time if the chain-agnostic validation failed. This can be optimized so that `_createMessageHash` is only called once, avoiding an unnecessary second call when the signer is not a Safe.

Impact

Informational.

Recommendation

Consider changing the `_processEIP1271Signature` so that only one call to `_createMessageHash` is done:

```
// File: SuperValidatorBase.sol

function _processEIP1271Signature(
        address contractSigner,
        SignatureData memory sigData
    )
        internal
        view
        returns (address)
    {
+       bytes32 messageHash = _createMessageHash(sigData.merkleRoot);
        // For Safe contracts: Try chain-agnostic validation first (cross-chain
        compatibility)
-       if (contractSigner.validateChainAgnosticMultisig(sigData,
        _createMessageHash(sigData.merkleRoot))) {
+       if (contractSigner.validateChainAgnosticMultisig(sigData, messageHash)) {
            return contractSigner;
        }

        // Generic EIP-1271 validation (works for ALL EIP-1271 contracts including
        Safe fallback)
-       bytes32 messageHash = _createMessageHash(sigData.merkleRoot);
        try IERC1271(contractSigner).isValidSignature(messageHash,
        sigData.signature) returns (bytes4 result) {
            if (result == IERC1271.isValidSignature.selector) {
                return contractSigner;
            }
        } catch { }

        revert NOT_EIP1271_SIGNER();
    }
```

Superform

Fixed in PR 746.

Orion

Fix confirmed.

I-02: CircleGatewayMinterHook does not reuse cached `asset` in `_postExecute` INFO

Vulnerability details

`CircleGatewayMinterHook.sol` sets `asset` in `_preExecute` to the decoded USDC token, but `_postExecute` re-decodes the token from the attestation payload by calling `_extractTokenFromAttestation(data)` instead of reusing `asset`. This adds unnecessary decoding part.

Impact

Informational

Recommendation

Use the cached `asset` set during `_preExecute` directly in `_postExecute` (and optionally assert `asset != address(0)` before use) rather than re-decoding from payload.

Superform

Fixed in PR 751.

Orion

Fix confirmed.

I-03: CircleGatewayMinterHook contains redundant validations already enforced internally INFO

Vulnerability details

- Redundant signature check: after calling `_decodeAttestationData(data)`, the code checks `if (signature.length == 0) revert INVALID_DATA_LENGTH();`. The internal decoder already validates signature presence, bounds, and minimum length (>= 65), so the post-check for zero length is unnecessary.
- Redundant token zero-address check: `_preExecute` reverts if `usdc == address(0)` immediately after calling `_extractTokenFromAttestation(data)`, but the extractor itself already reverts when the decoded token is zero. This results in duplicate validation paths.

Impact

Informational.

Recommendation

- Remove the `signature.length == 0` check following `_decodeAttestationData`.
- Remove the `usdc == address(0)` check in `_preExecute` and rely on `_extractTokenFromAttestation` to enforce non-zero token.

Superform

Fixed in PR 751.

Orion

Fix confirmed.

I-04: Some redundant checks in `SuperDestinationValidator` can be removed INFO

Vulnerability details

The `SuperDestinationValidator` performs some checks in `_decodeDestinationData` to sanitize `sender_` and `chainId`:

```
// File: SuperDestinationValidator.sol
function _decodeDestinationData(
        bytes memory destinationDataRaw,
        address sender_
    )
        private
        view
        returns (DestinationData memory)
    {
        (
            bytes memory callData,
            uint64 chainId,
            address decodedSender,
            address executor,
            address[] memory dstTokens,
            uint256[] memory intentAmounts
        ) = abi.decode(destinationDataRaw, (bytes, uint64, address, address, address[],
        uint256[]));
        if (sender_ != decodedSender) revert INVALID_SENDER();

        if (chainId != block.chainid) revert INVALID_CHAIN_ID();
        return DestinationData(callData, chainId, decodedSender, executor, dstTokens,
        intentAmounts);
    }
```

However, these checks are redundant, as they check the same data that is passed to the `_decodeDestinationData` function:

```
// SuperDestinationExecutor.sol

 function processBridgedExecution(
        address,
        address account,
        address[] memory dstTokens,
        uint256[] memory intentAmounts,
        bytes memory initData,
        bytes memory executorCalldata,
        bytes memory userSignatureData
    )
        external
        override
    {
        ...SNIP

        // --- Signature Validation ---
        // DestinationData encodes executor calldata, current chain id, account, current
        executor, destination tokens
        // and intent amounts
        bytes memory destinationData =
            abi.encode(executorCalldata, uint64(block.chainid), account, address(this),
        dstTokens, intentAmounts);
```

As shown in the snippet, the `destinationData` encodes `block.chainid` as the `chainId` param, and the `account` as the address passed as the account. This makes the checks `_decodeDestinationData` ineffective, as they are comparing the same values.

Impact

Informational.

Recommendation

Consider removing the redundant checks in `_decodeDestinationData`. This will help save gas and improve the code quality.

Superform

Fixed in PR 753.

Orion

Fix confirmed, the checks have been removed.

I-05: `MerklClaimRewardHook` doesn't charge fees INFO

Vulnerability details

The `MerklClaimRewardHook` is configured as a `NONACCOUNTING` hook, which leads to fees not being applied for rewards claimed via Merkl and leading to a loss of revenue for Superform.

Impact

Informational.

Recommendation

Since Merkl's `claim` logic allows claiming rewards for multiple tokens in the same transaction, the current fee charging logic in Superform won't allow to charge fees just like it's done in all the other hooks.

One possible way to fix this is by updating the logic in the `_buildHookExecutions` from the `MerklClaimRewardHook`, so that additional executions are added to charge a fee for each token present in the claim transaction:

```
// File: MerklClaimRewardHook.sol
function _buildHookExecutions(
        address,
        address account,
        bytes calldata data
    ) internal view override returns (Execution[] memory executions) {
        ClaimParams memory params;

        // decode users
        address[] memory users = _setUsersArray(account, data);
        params.users = users;

        // decode other params
        (params.tokens, params.amounts, params.proofs) = _decodeClaimParams(
            data
        );

-        executions = new Execution[](1);
+        executions = new Execution[](users.length + 1);
        executions[0] = Execution({
            target: distributor,
            value: 0,
            callData: abi.encodeCall(
                IDistributor.claim,
                (params.users, params.tokens, params.amounts, params.proofs)
            )
        });

+        for (uint256 i; i < users.length; ) {
+            (uint208 amount, , ) = IDistributor(distributor).claimed(
+                params.users[i],
+                params.tokens[i]
+            );

+            uint256 fee = ((params.amounts[i] - amount) * FEE_BPS) / BPS;

+            executions[i + 1] = Execution({
+                target: params.tokens[i],
+                value: 0,
+                callData: abi.encodeCall(IERC20.transfer, (feeReceiver, fee))
+            });
        }
    }
```

It is however important to highlight the fact that this approach has some limitations if compared with the fee logic in the executor contract: - Unlike `_performErc20FeeTransfer`, we can't verify deviations in the transfer, given that this approach won't actually execute the code until the handleOps execution - The `feeReceiver` must be configured in the hook, instead of being fetched for each token from a yield configuration

An alternative approach (which would require more changes) is to support multiple tokens when charging fees in the SuperExecutor.

Superform

Fixed in PR 770.

Orion

Fix confirmed. Fees are now charged in the `MerklClaimRewardHook` based on the amount being claimed in the current execution

I-06: EIP-2098 signatures are not supported in `BatchTransferFromHook`

INFO

Vulnerability details

Permit2 includes support for EIP-2098 signatures, as shown in it's signature verification library, whose length is 64 bytes.

However, the `BatchTransferFromHook` encoded data assumes signatures are always 65-byte long :

```
function _buildHookExecutions(
        address,
        address account,
        bytes calldata data
    )
        internal
        view
        override
        returns (Execution[] memory executions)
    {
       ...SNIP

        vars.signature = BytesLib.slice(data, data.length - 65, 65);

       ...SNIP
```

This will make it impossible to accept EIP-2098 signatures, despite them being valid.

Impact

Informational.

Recommendation

Consider updating the decoding approach so that both 65-byte and 64-byte signatures are accepted.

I-07: DeBridge's permit functionality can't be used when creating an order INFO

Vulnerability details

DeBridge allows users to provide a `_permitEnvelope` when creating a new order. This is a bytes parameter that is used to approve the spender through a signature via a permit call.

The problem is that the `DeBridgeSendOrderAndExecuteOnDstHook.sol` hardcodes this parameter to an empty bytes array, effectively preventing users from using this feature from DeBridge:

```
// File: DeBridgeSendOrderAndExecuteOnDstHook.sol
function _buildHookExecutions(
        address prevHook,
        address account,
        bytes calldata data
    )
        internal
        view
        override
        returns (Execution[] memory executions)
    {
        ...SNIP

        // build execution
        executions = new Execution[](1);
        executions[0] = Execution({
            target: dlnSource,
            value: value,
            callData: abi.encodeCall(IDlnSource.createOrder, (orderCreation,
        affiliateFee, referralCode, "")) <@
        });
    }
```

Impact

Informational. Users can still use SuperForm's permit hook and have a similar outcome to DeBridge's permit envelope, but it is important to highlight this limitation.

Recommendation

Consider allowing users to encode an additional `bytes` parameter in the `DeBridgeSendOrderAndExecuteOnDstHook`, allowing them to specify the permit envelope data.

Superform

Acknowledged. *"That is intended for the reason mentioned in the item. We perform the approval with our own and decided a while ago to strip the permit envelope from the params"*

Orion

Acknowledged.

I-08: `BaseLedger` uses concrete `SuperLedgerConfiguration` contract instead of `ISuperLedgerConfiguration` interface INFO

Vulnerability details

`BaseLedger.sol` stores `superLedgerConfiguration` as concrete `SuperLedgerConfiguration` type instead of `ISuperLedgerConfiguration` interface. This includes the entire SuperLedgerConfiguration bytecode (~320 lines) in BaseLedger deployment, unnecessarily increasing deployment costs and contract size.

Impact

Informational.

Recommendation

Use the interface type for storage:

```
ISuperLedgerConfiguration public immutable superLedgerConfiguration;
```

Superform

Fixed in PR 777.

Orion

Fix confirmed.

I-09: `previewFees` calculates inconsistent fees when shares exceed tracked amount INFO

Vulnerability details

`previewFees` function doesn't adjust `amountAssets` when `usedShares` are capped by `calculateCostBasisView`, unlike `_processOutflow` which recalculates `amountAssets` based on capped shares. This causes preview fees to be higher than actual fees when users have shares deposited outside Superform.

Impact

Informational.

Recommendation

Acknowledge this limitation in the NatSpec comment:

```
/// @notice Previews performance fees for a potential withdrawal
/// @dev This preview may show higher fees than actual withdrawal when usedShares
///      exceed tracked shares, as it doesn't adjust amountAssets like _processOutflow
     does
function previewFees(
    ...
) public view returns (uint256 feeAmount)
```

Superform

Acknowledged.

Orion

Acknowledged.

I-10: Fee calculations round down instead of up, favoring users over protocol INFO

Vulnerability details

Fee calculations in `BaseLedger.sol` use `Math.mulDiv` which rounds down, favoring users over the protocol in two places:

1. **Asset amount calculation**: When recalculating `amountAssets` from capped shares in `_processOutflow` function, rounding down reduces the asset value, leading to lower profit and lower fees
2. **Fee calculation**: When calculating fees from profit in `_calculateFees` function, rounding down directly reduces fee amount

Both rounding directions favor users over the protocol. While `Math.mulDivUp` and other rounding-up functions are available in the codebase (Solady's FixedPointMathLib), the current implementation uses the standard rounding-down approach.

For most tokens with 18 decimals, the impact is negligible. However, for tokens like BTC with 8 decimals at $120k price, the maximum rounding loss per operation can be around a few cents based on the fee percentage.

Impact

Informational.

Recommendation

Consider rounding-up to favor the protocol:

```
// For asset calculation
amountAssets = Math.mulDiv(updatedUsedShares, pps, 10 ** decimals, Math.Rounding.Ceil);

// For fee calculation
feeAmount = Math.mulDiv(profit, feePercent, 10_000, Math.Rounding.Ceil);
```

Superform

Acknowledged.

Orion

Acknowledged by the Superform team.

I-11: Manager can bypass gradual fee increase limit by setting fee to 0 first INFO

Vulnerability details

The `proposeYieldSourceOracleConfig` function has a vulnerability that allows managers to bypass the gradual fee increase protection. The fee change validation logic only applies when `existingConfig.feePercent > 0`, but has no restrictions when the existing fee is 0.

This creates a bypass mechanism: 1. Manager sets fee to 0% (allowed from any percentage) 2. After timelock period, accepts the proposal (fee becomes 0%) 3. Manager immediately proposes new fee up to 50% (MAX_FEE_PERCENT) 4. After timelock, accepts the proposal

The gradual increase limit (`MAX_FEE_PERCENT_CHANGE = 5000` basis points = 50%) is completely bypassed, allowing instant jumps from 0% to 50% fees.

Impact

Informational.

Recommendation

Add a maximum fee increase limit when transitioning from 0% fees:

```
if (existingConfig.feePercent > 0) {
    // existing validation logic
} else if (existingConfig.feePercent == 0 && config.feePercent > 0) {
    // Limit initial fee setting to 5-10%
    uint256 MAX_INITIAL_FEE_PERCENT = 1000; // 10% in basis points
    if (config.feePercent > MAX_INITIAL_FEE_PERCENT) revert INVALID_FEE_PERCENT();
}
```

Superform

Fixed in PR 777.

Orion

Fix confirmed. When the `feePercent` in the existing config is 0, a maximum of `MAX_INITIAL_FEE_PERCENT` can be configured.

I-12: Fee change limits use incorrect rounding direction for `minFee` calculation INFO

Vulnerability details

The fee change validation in `proposeYieldSourceOracleConfig` uses inconsistent rounding directions for minimum and maximum fee calculations.

Both calculations use `Math.mulDiv` which rounds down, but this creates asymmetric rounding behavior that favors one direction over the other.

The issue is that **minFee should round up** to be more restrictive (higher minimum), while **maxFee correctly rounds down** to be more restrictive (lower maximum). Rounding down makes the minFee validation less restrictive than intended.

Impact

Informational.

Recommendation

Use appropriate rounding directions for each calculation:

```
uint256 minFee = Math.mulDiv(existingConfig.feePercent, (10_000 -
        MAX_FEE_PERCENT_CHANGE), 10_000, Math.Rounding.Ceil);
uint256 maxFee = Math.mulDiv(existingConfig.feePercent, (10_000 +
        MAX_FEE_PERCENT_CHANGE), 10_000); // Keep rounding down
```

Superform

Fixed in PR 777.

Orion

Fix confirmed. `minFee` is now computed by rounding up.

I-13: Missing array length validations in `SuperYieldSourceOracle` functions INFO

Vulnerability details

The `SuperYieldSourceOracle.sol` contract has multiple functions with incomplete array length validation, which can lead to out-of-bounds access.

Issues in GENERALIZED QUOTING FUNCTIONS section:

1. **`getPricePerShareMultipleQuote`**: Missing `oracles.length` validation
2. **`getTVLByOwnerOfSharesMultipleQuote`**: Missing `oracles.length` validation
3. **`getTVLMultipleQuote`**: Missing `oracles.length` validation

Issues in YIELD SOURCE ORACLE FUNCTIONS section:

4. **`getPricePerShareMultiple`**: No length validation at all
5. **`getTVLByOwnerOfSharesMultiple`**: No length validation at all
6. **`getTVLMultiple`**: No length validation at all

These missing validations can cause array out-of-bounds access when the arrays have mismatched lengths.

Impact

Informational.

Recommendation

Update length validations.

Superform

Fixed in PR 777.

Orion

Fix confirmed. All corresponding length validations have been added.

I-14: Some configured deployment addresses are incorrect INFO

Vulnerabiliy description

The `Constants.sol` contract contains a list of all the addresses that will be needed as configuration during deployment. For Spectra, these addresses are mentioned:

```
// File: Constants.sol

    ...SNIP

    address internal constant SPECTRA_ROUTER_MAINNET =
        0xD733e545C65d539f588d7c3793147B497403F0d2;
    address internal constant SPECTRA_ROUTER_BASE =
        0x0FC2fbd3E8391744426C8bE5228b668481C59532;
    address internal constant SPECTRA_ROUTER_OPTIMISM =
        0x7dcDeA738C2765398BaF66e4DbBcD2769F4C00Dc;
    address internal constant SPECTRA_ROUTER_ARBITRUM = address(0); // TODO: Research
        Arbitrum Spectra router address
    address internal constant SPECTRA_ROUTER_BNB = address(0); // TODO: Research BNB
        Spectra router address

    ...SNIP
```

For Morpho, only one address is configured:

```
// File: Constants.sol

    ...SNIP

    address public constant MORPHO = 0xBBBBBbbBBb9cC5e90e3b3Af64bdAF62C37EEFFCb;

    ...SNIP
```

Following Spectra's official documentation, we've identified that new contracts are available, making the hardcoded ones outdated. In addition, the `SPECTRA_ROUTER_ARBITRUM` and `SPECTRA_ROUTER_BNB` should be configured, as they are currently set to zero.

For Morpho, the configured address is only valid for Mainnet and Base. As per Morpho's official documentation, all other chains will differ.

Impact

Informational

Recommendation

Consider updating the Spectra Router addresses hardcoded in `Constants.sol` to reflect the latest deployed versions.

Superform

Fixed in PR 777 and PR 785.

Orion

Fix confirmed.

I-15: Incorrect variable used in DebridgeAdapter verification in verify_v2_prod.sh INFO

Vulnerability details

The `verify_v2_prod.sh` script uses the wrong variable `debridge_dst_dln` instead of the correct `debridge_dln_dst` variable in the DebridgeAdapter verification process, which will cause verification failures on Tenderly.

1. The `debridge_dst_dln` variable is declared and left empty

   ```
   local debridge_dst_dln=""
   ```

2. The `debridge_dst_dln` variable is used in constructor encoding for DebridgeAdapter instead of `debridge_dln_dst`

   ```
   echo "$(cast abi-encode "constructor(address,address)" "$debridge_dst_dln"
           "$super_destination_executor")"
   ```

This will result in an empty/invalid constructor argument being passed to the DebridgeAdapter contract during Tenderly verification, causing the verification process to fail.

Impact

Informational.

Recommendation

Replace `debridge_dst_dln` with `debridge_dln_dst` in the constructor encoding to use the correct variable that contains the actual DLN destination address, and remove the unused `debridge_dst_dln` variable from the script.

Superform

Fixed in PR 782.

Orion

Fix confirmed.